UHC user manual, version 1.1.4

Atze Dijkstra

September 28, 2012

Contents

1 Introduction and installation **2**
1.1 System overview . . . 2
1.2 Download, build, install, compile and run . . . 2
1.2.1 Download . . . 2
1.2.2 Build and install UHC . . . 3
1.2.3 Run UHC . . . 3
1.2.4 Test UHC . . . 3
1.2.5 Troubleshoot . . . 4
1.2.6 More make commands . . . 4
1.2.7 Configuration parameters . . . 5

2 Using UHC **5**
2.1 Commandline invocation . . . 5
2.2 Commandline options . . . 6
2.3 Cabal and packages . . . 8
2.4 Optimizations . . . 8

3 Language extensions **8**
3.1 Higher ranked types and impredicativity . . . 9
3.1.1 Higher ranked types . . . 9
3.1.2 Co- and contravariance . . . 9
3.1.3 Impredicativity . . . 10
3.1.4 Caveats . . . 11
3.2 Standalone deriving instance . . . 11
3.3 Generic deriving . . . 11
3.4 Partial type signature . . . 12
3.5 Quantifier position inference . . . 13
3.6 Existential types . . . 13
3.7 Kind inference and signatures . . . 14
3.8 Local instances . . . 14
3.9 Lexically scoped type variables . . . 15
3.10 Extensible records . . . 15
3.11 Relaxed monomorphism restriction . . . 16
3.12 Default for ambiguous overloading . . . 17
3.13 Infix type constructors, classes, and type variables . . . 17
3.14 Pragmas . . . 17
3.15 Preprocessing (with `cpp`) . . . 18

4 Haskell compatibility **18**
4.1 Haskell 98 . . . 18
4.2 Haskell 2010 . . . 19

5 Backend specifics **19**
5.1 Foreign function interface (FFI) . . . 19
5.1.1 Primitives . . . 19
5.1.2 `bc` and `C` backend . . . 19
5.1.3 `js` backend . . . 19
5.2 Limitations . . . 21

6 Language implementation status **21**
6.1 Included packages . . . 21
6.2 Library modules . . . 22
6.3 Library + runtime . . . 22
6.3.1 Third party libraries . . . 22
6.4 Known issues . . . 23
6.5 Fully functional backends . . . 23
6.5.1 `bc`: GRIN based interpreter, no whole program analysis . . . 23
6.6 Almost fully functional backends . . . 23
6.6.1 `C`: GRIN based C, with whole program analysis . . . 23
6.6.2 `jazy`: Core based Java, no whole program analysis . . . 24
6.6.3 `js`: Core based JavaScript, no whole program analysis . . . 24
6.7 Partially functional backends . . . 24
6.7.1 `llvm`: GRIN based LLVM, with whole program analysis . . . 24
6.7.2 `clr`: GRIN based CLR, with whole program analysis . . . 24

7 Troubleshooting FAQ **24**
7.1 The compiler seems to loop, or complains about premature end of file, or complains with "ehc: Prelude.chr: bad argument: 4087767" . . . 24
7.2 I have installed UHC previously, now the libraries do not compile . . . 24
7.3 I have checked out a new EHC version over a previously working one, and now it does not compile anymore . . . 25

8 License **25**

9 Further reading **26**

1 Introduction and installation

For version 1.1.4

1.1 System overview

For now, see the UHC structure overview.

1.2 Download, build, install, compile and run

A separate manual exists on how to install UHC on a Windows system. A list of often occurring build problems is maintained here.

1.2.1 Download

- Prerequisites. Running the configure scripts yields an overview of what is missing.

  - GHC: a recent version, preferably `>= 7.0.3`; `GHC 7.0.3` has been used for recent development, `GHC 7.4.1` for current development. Older GHC versions may do as well, but are not used for developing, nor is any effort done for keeping `UHC` compilable with older `GHC` versions. The installed libraries should include the `mtl` and `fgl` library. Depending on platform and `GHC` distribution more libraries may need to be installed.
  - Additional libraries, available via Hackage or locally.
    * uulib: parser combinators.
    * uuagc: attribute grammar system.
- Download
  - See the download page.

1.2.2 Build and install UHC

- Prerequisites: GHC and the abovementioned libraries must be installed.
  - Look here for setting up a cygwin environment and additional info on installing UHC under cygwin.
- For building UHC, run the following commands from trunk/EHC:

```
% ./configure
% make uhc                          # under the hood, this is EHC variant 101
% make install                      # sudo may be required, depending on your permissions
```

  During the build part of the compiler is built as a library, installed as a ghc user package, via cabal.

1.2.3 Run UHC

- Run UHC. A sample session:

```
% cat > hw.hs
module HelloWorld where
main = putStrLn "Hello World"
% uhc hw.hs
[1/1] Compiling Haskell              hw                     (hw.hs)
% ./hw
Hello World
%
```

By default an executable for code generation target `bc` (bytecode interpreter) is generated.

1.2.4 Test UHC

- Regress test UHC:

```
% make test
WARNING: output may differ w.r.t. names for the following tests:  IO2.hs IO3.hs
make test-regress TEST_VARIANTS=uhc
== version 101 ==
  -- 99/BoundedChar1.hs.exp101 -- 99/BoundedChar1.hs.reg101 --
  -- 99/BoundedInt1.hs.exp101 -- 99/BoundedInt1.hs.reg101 --
  -- 99/CaseFall1.hs.exp101 -- 99/CaseFall1.hs.reg101 --
...
%
```

  Differences may also be reported because of platform dependent representation of linefeeds. The tests where this is likely to happen are marked with the word *platform* to indicate the platform or runtime environment differences.

1.2.5 Troubleshoot

See the troubleshooting FAQ (page 24).

1.2.6 More make commands

Type `make help` to see what more can be build:

```
UHC
===
make                    : defaults to 'make uhc'
make uhc                : make uhc and library (ehc variant 101)
make install            : make uhc and install globally (into /usr/local/lib/uhc-1.1.4/), possibly needing admin permi
make uninstall          : uninstall uhc, possibly needing admin permission
make test               : regress test uhc

EHC & tools
===========
make <n>/ehc            : make compiler variant <n> (in bin/, where <n> in {1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 18
make <n>/ehclib         : make ehc library (i.e. used to compile with ehc) variant <n> (in bin/, where <n> in {99 100
make <n>/ehclibs        : make ehc libraries for all codegen targets
make <n>/rts            : make only the rts part of a library
make <n>/bare           : make bare source dir for variant <n> (in bare/),
                          then 'cd' to there and 'make'
make ruler              : make ruler tool
make shuffle            : make shuffle tool

Documentation
=============
make help               : print this help
make www                : make www documentation: doc/shuffle-doc.pdf doc/text2text-doc.pdf doc/howtodoc-doc.pdf doc/h
make www-sync           : install www documentation in the EHC web (http://www.cs.uu.nl/wiki/Ehc/WebHome)

make doc/<d>.pdf        : make (public) documentation <d> (where <d> in { ruler-doc ehc-book ehc-doc}),
                          or (non-public): <d> in { flops06-ruler-paper flops06-ruler popl07-explimpl hw06-impred esop
                          or (doc): <d> in { shuffle-doc text2text-doc howtodoc-doc howtoexperiment-doc ehc-technical-
                          only if text src available, otherwise already generated

Testing
===========
make test-regress       : run regression test,
                          restrict to versions <v> by specifying 'TEST_VARIANTS=<v>' (default '1 2 3 4 5 6 7 8 9 10 11
                          requires corresponding ehc/grini/ehclib already built
make test-expect        : make expected output (for later comparison with test-regress), see test-regress for remarks
make benchmark          : run 16 nofib programs with 3 compilers on 3 inputs each

Distribution
===========
make uhc-dist           : make platform specific binary distribution from uhc build in /Volumes/Work/Programming/uhc/E

Cleaning up
===========
make <n>/clean          : cleanup for variant <n>
make clean              : cleanup all variants + internal libraries and tools
make clean-extlibs      : cleanup external libraries

Other
=====
make ehcs               : make all compiler (ehc) versions
make top                : make Typing Our Programs library
make lvm                : make Lazy Virtual Machine library
make helium             : make Helium library
make heliumdoc          : make Haddock documentation for Helium, Top and Lvm (in hdoc/)

Obsolesence candidates
======================
```

```
make <n>/infer2pass      : make infer2pass demo version <n> (in bin/, where <n> in {1 2 3})
make <n>/grini           : make grin interpreter variant <n> (in bin/, where <n> in {8 9 10 11 12 13 14 15 17 18 19 20
make <n>/hdoc            : make Haddock documentation for variant <n> (in hdoc/)
make grinis              : make all grin interpreter (grini) versions
```

1.2.7 Configuration parameters

Apart from the usual options the `./configure` accepts the following options enabling a particular feature. Unless mentioned otherwise, the default is `no`.

- `--enable-java`. Enable `jazy` backend.
- `--enable-jscript`. Enable `jscript` (Javascript) backend. Default=`yes`.
- `--enable-llvm` (implies `wholeprogAnal`, `wholeprogC`). Enable `llvm` backend.
- `--enable-clr` (implies `wholeprogAnal`, `wholeprogC`). Enable `clr` backend.
- `--enable-tycore`. Enable `TyCore` typed core intermediate representation.
- `--enable-tauphi` (implies `tycore`). Enable `TyCore` based experimental strictness optimizations.
- `--enable-wholeprogC` (implies `wholeprogAnal`). Enable the `C` whole program analysis backend.
- `--enable-wholeprogAnal`. Enable whole program analysis.
- `--enable-coresysf` (under construction). Enable System F generation for Core.
- `--enable-cmm` (under construction). Enable Cmm intermediate language for C code generation.

Replacing `enable` with `disable` will have the reverse effect.

2 Using UHC

2.1 Commandline invocation

As printed by `install/100/bin/ehc --help`:

```
version: ehc-1.1.4, revision master@1cb861b398, timestamp 20120928 +0200 144218, aspects: base hmtyinfer codegen grin n

Usage: ehc [options] [file[.eh|.hs] ...]

options:
  -h                        --help                              print this help (then stop)
                            --version                           print version info (then stop)
                            --version-dotted                    print version in "x.y.z" style (then stop)
                            --version-asnumber                  print version in "xyz" style (then stop)
                            --numeric-version                   see --version-dotted (to become obsolete)
  -v[0|1|2|3|4]             --verbose[=0|1|2|3|4]               be verbose, 0=quiet, 4=debug, default=1
  -t bc|jazy|js             --target=bc|jazy|js                 generate code for target, default=bc
                            --target-flavor=debug|plain         generate code for target flavor, default=plain
  -p[hs|eh|ast|-]           --pretty[=hs|eh|ast|-]              show pretty printed source or EH abstract syntax tree, d
  -O[0|1|2|3|<opt>[=Bool]]  --optimise[=0|1|2|3|<opt>[=Bool]]   optimise with level or specific by name, default=1
                            --no-recomp                         turn off recompilation check (force recompile)
                            --no-prelude                        do not assume presence of Prelude
                            --no-hi-check                       no check on .hi files not matching the compiler version
  -c                        --compile-only                      compile only, do not link
  -i path                   --import-path=path                  search path for user files, separators=';', appended to
```

```
-L path                 --lib-search-path=path            search path for library files, see also --import-path
                        --cpp                             preprocess source with CPP
                        --limit-tysyn-expand=<nr>         type synonym expansion limit
                        --odir=dir                        base directory for generated files. Implies --compile-on
-o file                 --output=file                     file to generate executable to
                        --keep-intermediate-files         keep intermediate files (default=off)
                        --meta-variant                    meta: print variant (then stop)
                        --meta-target-default             meta: print the default codegeneration target (then stop
                        --meta-targets                    meta: print supported codegeneration targets (then stop)
                        --meta-optimizations              meta: print optimization names (then stop)
                        --meta-pkgdir-system              meta: print system package dir (then stop)
                        --meta-pkgdir-user                meta: print user package dir (then stop)
                        --package=package                 see --pkg-expose
                        --hide-all-packages               see --pkg-hide-all
                        --pkg-build=package               pkg: build package from files. Implies --compile-only
                        --pkg-expose=package              pkg: expose/use package
                        --pkg-hide=package                pkg: hide package
                        --pkg-hide-all                    pkg: hide all (implicitly) assumed/used packages
                        --pkg-searchpath=path             pkg: package search directories, each dir has <pkg>/<var
                        --cfg-install-root=dir            cfg: installation root (to be used only by wrapper scrip
                        --cfg-install-variant=variant     cfg: installation variant (to be used only by wrapper sc
                        --optP=opt for cmd                opt: option for cmd used by compiler, currently only P (
                        --coreopt=opt[,...]               core opts: pp-parseable
```

2.2 Commandline options

Only options offered by UHC are discussed, lower variants of UHC have more options for debugging available. Other options can be used but are either for internal use of can be disabled sometime. Defaults for options are valid only for variant 100 of EHC. Boolean options are indicated by `Bool`, where `True` and `False` can be indicated by one of `1|yes|on|+` respectively `0|no|off|-`.

- `-h`, `--help`. Print commandline help and quit.
- `--version`. Print a longer version description.
- `--version-dotted`. Print the version in a dotted format, as used by many version aware utilities (like cabal)..
- `--version-asnumber`. Same as `--version-dotted` but without the dots, the value `__UHC__` has when preprocessing with cpp.
- `--numeric-version`. See `--version-dotted`.
- `-v[<level>]`, `--verbose[=<level>]`. Verbosity level. Verbosity `<level>`s are accumulative in their reported messages:
  - `0`: quiet.
  - `1` (default): minimal report of compile phases
  - `2`: report of compile phases, including import handling (and similar info).
  - `3`: progress of internal processing.
  - `4`: additional debugging info.
- `-t[<target>]`, `--target[=<target>]`. Generate code for a target. Code generation `<target>`s (Default: bc):
  - `bc`: bytecode interpreter based executable. See `bc` backend (page 23).
  - `C`: whole program analysis C code based executable. See `C` backend (page 23).

- `--target-flavor[=<flavor>]`. Generate code for a target. Code generation `<flavor>`s (Default: plain). Flavors distinguish between incompatible ways of generating code. This is not yet used, except for experimenting with code generation with extra debugging, in particular stack tracing.
  - `plain`: nothing special.
  - `debug`: debugging experiments.
- `-O[<level>][,<scope>]`, `--optimise[=<level>][,<scope>]`. Optimisation level and scope. Currently this makes little difference as few optimisations are implemented. Levels:
  - `0`: none.
  - `1` (default): basic.
  - `2`: much
  - `3`: all
  - `<opt>[=Bool]`: turn on/off optimization option `<opt>` on top of those already defined. See also `--meta-optimizations` optimizations (page 8).

  The scope determines at which point the optimizations are done, by default on a per module basis, optionally on a whole program by linking in an earlier compiler pipeline stage. The linking then only pulls in that what is required, and can do the cheaper optimisations on the full program as well.
  - `0`: per module (default).
  - `1`: link per module GRIN to whole program GRIN, and continue from there (not yet implemented)
  - `2`: link per module Core to whole program COre, and continue from ther; Core precedes GRIN in the compiler pipeline.
- `--no-recomp`. Don't check for the necessity to recompile, recompile allways instead.
- `--no-prelude`. Don't assume the presence of `Prelude`.
- `--no-hi-check`. Don't use out of date of compiler version w.r.t. to .hi files as a reason to recompile. This is useful when debugging the compiler if you know that .hi files will not change. Use this option only if you know what you are doing.
- `--cpp`. Preprocess with cpp. See also preprocessing (page 18).
- `--limit-tysyn-expand=<nr>`. Limit the level of type synonym expansion.
- `-i<path>`, `--import-path=<path>`. Add `<path>` to the search path for importing modules and cpp.
- `--odir=dir`. Generated files are put in `dir`. More precisely, for each generated artefact for a module, the module name with dots `'.'` replaced by slashes `'/'` is appended to `dir` and used as the actual base name. When compiling for the construction of a package, an additional 4 level directory structure is prefixed, encoding package name, compiler version, target, and flavor.
- `--keep-intermediate-files`. Keep intermediate files (such as `.c` files and output of cpp).
- `--meta` options. These are mostly used to let makefiles know what can be done.
  - `--meta-optimizations`: names of all possible optimizations. See also optimizations (page 8).
  - `--meta-targets`: all possible code generation targets.
  - `--meta-target-default`: default code generation target.

  - `--meta-variant`: EHC variant.
  - `--meta-pkgdir-system`: Print the builtin system dir holding packages, and quit.
  - `--meta-pkgdir-user`: Same as `--meta-pkgdir-system`, but user dir.

- `--pkg` options. These are currently used as a simplistic equivalent of ghc-pkg, to be used only internally by library construction. Expect changes here.
  - `--pkg-build=<pkg>`: build `<pkg>` from generated files. Put in current directory of the one specified with `--odir`.
  - `--pkg-expose=<pkg>`: expose `<pkg>` to be visible so its modules can be imported.
  - `--pkg-hide=<pkg>`: reverse of `--pkg-expose`.
  - `--pkg-hide-all`: hide all packages.
  - `--pkg-searchpath=path`: additional locations to search for packages.
- `--opt<X>=opt`. Pass option to command for pass `<X>`. Currently only `P` is implemented, that is, passing to the preprocessor (CPP).

2.3 Cabal and packages

Cabal (version 1.9.3. and higher) has support for building libraries with UHC, using the flag `--uhc` to `cabal`. Building executables is not yet supported. UHC's package mechanism is accessed via commandline flags for exposing (`--pkg-expose`) and hiding (`--pkg-hide`) packages. Package identifiers may include a version number. Be aware of the following current limitations:

- Package disambiguation (i.e. a module resides in multiple packages) is done by using the version number: higher versions of the same package take precedence over lower versions, an unversioned package takes precedence over a versioned one, packages available earlier in the package search path take precedence. No further disambiguation is done: e.g. module `X` residing in both package `yy` and `zz` will cause unspecified behavior. There is no mechanism for specifying package names when importing.
- Only the default backend is supported. This is a limitation of cabal which does not yet have builtin infrastructure for dealing with different backends and variants of backends.

Some missing language features however may stand in the way of faultless library compilation, but all should be well if packages only assume package `haskell98` (part of the distribution). Keep in mind that cabal support for this release is new, largely untested, and works only for libraries. Executables cannot yet be build with cabal.

2.4 Optimizations

Currently few optimizations are implemented, apart from the work on whole program analysis. Optimizations can be individually turned on and off:

- `GrinLocal`: non whole program analysis grin optimizations to remove the most obvious inefficiencies.
- `StrictnessAnalysis` (in development): strictness analysis based on relevance analysis.

3 Language extensions

Bear in mind that these extensions are all but tried and tested!

3.1 Higher ranked types and impredicativity

3.1.1 Higher ranked types

Higher ranked types are specified explicitly:

```
poly1 :: (forall a . a -> a) -> (Int,Bool)
poly1 f = (f 1, f True)
```

Alternatively, the type information can be specified without a separate type signature by mixing annotations throughout patterns and expressions:

```
poly2 (f :: forall a . a -> a) = (f (1::Int), f True)
```

In both cases the relevant specified type information is propagated to where it is needed.
Such type information may be freely mixed with tuples:

```
poly3 :: (b,forall a . a -> a) -> (Int,b)
poly3 (b,f) = (f 1, f b)

poly4 (b,f :: forall a . a -> a) = (f (1::Int), f b)
```

or type constructors:

```
data T a b = T a b

poly5 :: T b (forall a . a -> a) -> (Int,b)
poly5 (T b f) = (f 1, f b)

poly6 (T b (f :: forall a . a -> a)) = (f (1::Int), f b)
```

Quantified types may also be used at arbitrary rank, in this case rank-3:

```
poly7 :: ((forall a . a -> a) -> (Int,Bool)) -> (Int,Bool)
poly7 poly = poly id

pval = poly7 poly1
```

3.1.2 Co- and contravariance

With rank co- and contravariance swap. For example, in the following a more general function can be used where a less general is expected; `id` can be used as an implementation for `ii`:

```
ii :: Int -> Int
ii = id
```

However, the following does *not* work because the instantiation relation swaps direction on a contravariance position.

```
id2 :: (forall a . a -> a) -> Int
id2 i = i 3

ii2 :: (Int -> Int) -> Int
ii2 = id2                       -- type error
```

`ii2` cannot pass to `id2` a function of type `forall a . a -> a` because it only gets a `Int -> Int`! However, the other way around is perfectly ok:

```
id2 :: (forall a . a -> a) -> Int
id2 = ii2

ii2 :: (Int -> Int) -> Int
ii2 i = i 3
```

This extends to higher ranks as well:

```
id3 :: ((forall a . a -> a) -> Int) -> Int
id3 i = i id

ii3 :: ((Int -> Int) -> Int) -> Int
ii3 = id3
```

`id3` gets a function to which it must pass a function of type `forall a . a -> a`; it is then ok when this function is used as `Int -> Int`, so as a value for `ii3` we can use `id3`. Both of these invocations are therefore legal:

```
id3app1 = id3 id2
id3app2 = id3 ii2
```

However, the other way around fails for passing `ii3` a function `id2` which then *must* be given a `forall a . a -> a`.

```
ii3app1 = ii3 id2        -- type error
ii3app2 = ii3 ii2
```

The mechanism extends to the use of class predicates:

```
ieq  :: Eq a => a -> a
ieq = id

iord  :: Ord a => a -> a
iord = ieq
```

Similarly for rank 2 we can define:

```
ieq2  :: (forall a . Eq a => a -> a) -> Int
ieq2 = iord2

iord2  :: (forall a . Ord a => a -> a) -> Int
iord2 i = i 3
```

This is ok: `ieq2` constructs a function accepting an `Ord` dictionary from the function it gets passed itself, which can then be passed to `iord2`. The compiler provides this coercion automatically.

Currently this mechanism has limits:

- Although co- and contravariance is propagated and analysed through datatypes, coercions are only derived for functions and tuples.

3.1.3 Impredicativity

UHC allows impredicativity, albeit with assistance of a programmer. For example, in the following example impredicativity means that the type of the argument to `single` propagates with quantifiers; this must be done explicitly by using in front of an argument:

```
single :: forall a . a -> [a]
single x = [x]

ids1 = single  id          -- :: forall a . [a->a]
ids2 = single ~id          -- :: [forall a . a->a]
```

For `ids2` the type `forall a . a->a` has been propagated via the argument type variable of `single` to inside the list. For `ids1` the argument is first instantiated.

Alternatively we can enforce this by providing a type signature:

```
ids3 = single  id :: [forall a . a->a]
ids4 = single (id :: forall a . a->a)
ids5 = (single :: (forall a. a-> a) -> [forall a. a->a]) id
```

All these result in the same type as for `ids2`. Without type signature or use of the type of the argument will not be propagated impredicatively.

3.1.4 Caveats

- Inference is order sensitive, and sometimes difficult to predict/characterize:

```
cons1  = (:) (\x->x)  ids
cons2  = (:) (\x->x) ~ids                                        -- should be: [forall a. a->a], but is fo
cons2' = (:) (\x->x :: forall a . a -> a) ~ids        -- the extra annotation fixes this
```

- Requires programmer assistance to remedy this.

3.2 Standalone deriving instance

Deriving clauses for a datatype can be specified independent of a datatype definition. The notation follows GHC, hence see GHC's documentation on Stand-alone deriving declarations.

3.3 Generic deriving

Instance deriving can be generically specified. The mechanism consists of three ingredients: specifying deriving behavior, binding the behavior to an instance, and declaring an instance. The latter -declaring an instance- is done as usual, by a deriving clause, either as part of a datatype definition or as a standalone clause. Currently generic deriving is used for classes `Eq`, `Bounded`, `Functor`, and `Typeable` in the base libraries.

As an example the code for `Bounded` from the `Prelude` is shown here; the details, datatypes, and limitations are explained in the paper A generic deriving mechanism for Haskell.

As is practice with generic programming, programming is done in terms of sum, product, and unit types:

```
data (:+:) f g p = L1 { unL1 :: f p } | R1 { unR1 :: g p }  -- sum
data (:*:) f g p = f p :*: g p                              -- product
data U1 p = U1                                              -- unit for products
newtype K1 i c p = K1 { unK1 :: c }                         -- wrapper for type constants (over which no deriving is do
newtype M1 i c f p = M1 { unM1 :: f p }                     -- wrapper for meta information (constructor, ...)
```

For each datatype instance of `Representable0` are generated by the compiler, used to convert between the datatype and the generic representation:

```
-- | Representable types of kind *
class Representable0 a rep where
  -- | Convert from the datatype to its representation
  from0  :: a -> rep x
  -- | Convert from the representation to the datatype
  to0    :: rep x -> a
```

The generic programmer is required to do the following:

- `Bounded'` **helper class**. Generic behavior is defined for a helper class, later bound to the real class.

```
class Bounded' f where
  minBound' :: f x
  maxBound' :: f x
```

- **Specifying deriving behavior for** `Bounded'`.

```
instance Bounded' U1 where
  minBound' = U1
  maxBound' = U1

instance (Bounded' fT) => Bounded' (M1 iT cT fT) where
  minBound' = M1 minBound'
  maxBound' = M1 maxBound'

instance (Bounded' fT, Bounded' gT) => Bounded' (fT :*: gT) where
  minBound' = minBound' :*: minBound'
  maxBound' = maxBound' :*: maxBound'

instance (Bounded fT) => Bounded' (K1 iT fT) where
  minBound' = K1 minBound
  maxBound' = K1 maxBound

instance (Bounded' fT, Bounded' gT) => Bounded' (fT :+: gT) where
  minBound' = L1 minBound'
  maxBound' = R1 maxBound'
```

- **Binding the behavior to derived instances**. Each generic function requires a `Representable0` instance and generic behavior (`Bounded'` helper), which are tied together with a generic function (e.g. `minBoundDefault`) and bound to the derivable function with a `DERIVABLE` language pragma.

```
{-# DERIVABLE Bounded minBound minBoundDefault #-}
minBoundDefault  ::  (Representable0 aT repT, Bounded' repT)
                     =>  repT xT -> aT
minBoundDefault rep = to0 (minBound' `asTypeOf` rep)

{-# DERIVABLE Bounded maxBound maxBoundDefault #-}
maxBoundDefault  ::  (Representable0 aT repT, Bounded' repT)
                     =>  repT xT -> aT
maxBoundDefault rep = to0 (maxBound' `asTypeOf` rep)
```

3.4 Partial type signature

Partial type signatures are fully specified type signatures where parts may be omitted and then later filled in by the type inferencer. Partial type signatures offer a middle way between

- Fully specifying a signature. This can be difficult for complex types.
- Not specifying a signature at all. The type inferencer may then not be able to infer a type.

The idea is to provide just enough information so that the type inferencer can figure out the rest. This is convenient when a signature involves higher ranked types, which the type inferencer can not infer on its own:

```
f1 :: (forall a . a->a) -> ...
f1 i = (i 'a', i True)
```

The dots `...` stand for a *wildcard*, the part to be filled in by the type inferencer, which then infers type `(forall a . a -> a) -> (Char,Bool)`.

A wildcard can also be given a name like any type variable in a type expression. To differentiate it from normal type variables the prefix `%` is used, which indicates that the type expression should not be quantified over this type variable:

```
f1 :: (forall a . a->a) -> %b
```

Giving a name to a wildcard is only useful when referred to from another part of the signature:

```
f1 :: (forall a . a->a) -> (%b,%b)
f1 i = (i 'a', i True)
```

Both elements of the tuple are enforced to have the same type, in the example this leads to a type error.

3.5 Quantifier position inference

If left unspecified, the position of a quantifier not automatically is placed in front of a type. For example, the type signature

```
unsafeCoerce :: a -> b
```

is interpreted to be:

```
unsafeCoerce :: forall a . a -> forall b . b
```

The idea is that after passing a parameter, the result type still can be chosen.

[more explanation here...]

The combination with impredicativity can sometimes cause unexpected behavior.

[more explanation here...]

3.6 Existential types

Types can be hidden using existential types, using the keyword `exists`:

```
x1 :: exists a . a
x1 = 3 :: Int
```

`x1` holds an `Int` but when `x1` is used this knowledge is forgotten. As a consequence nothing can be done anymore with `x1`, except passing it to polymorphic functions (like `id`) which do not assume anything about the value being passed.

More useful is the combination with a function accepting a value of the hidden type, providing encapsulation and data abstraction:

```
x2 :: exists a . (a, a -> Int)
x2 = (3 :: Int, id)

xapp :: (exists b . (b,b -> a)) -> a
xapp (v,f) = f v

x2app = xapp x2
```

An existential type being bound to a value identifier has meaning, in that it fixates the type for the existential. The type of `x2` is some type invented by the compiler and cannot be named by the programmer. This is to prevent illegal mixing up two different existentials, while still be able to compute different existentials depending on some input:

```
mkx :: Bool -> exists a . (a, a -> Int)
mkx b = if b then x2 else ('a',ord)

y1 = mkx True       -- y1 :: (C_3_225_0_0,C_3_225_0_0 -> Int)
y2 = mkx False      -- y2 :: (C_3_245_0_0,C_3_245_0_0 -> Int)

mixy = let (v1,f1) = y1
           (v2,f2) = y2
       in  f1 v2
```

`mixy` causes a type error. However, we can use `y1` and `y2` perfectly well:

```
main :: IO ()
main
  = do putStrLn (show (xapp y1))
       putStrLn (show (xapp y2))
```

The invented types for bound existentials are allowed to be used everywhere; their use is limited by the availability of functions existentially hidden together with the existential type.

Existential types can even be used in a manner parallel to laziness on the value level:

```
f :: forall a . (a -> exists b . (b, a, b -> b -> Int))
f i = (3, i, (+))

main = print (let  (b, a, g) = f b
              in   g b a)
```

The `b` and its type are known after the invocation of `f` but is immediately passed back as an argument to `f`, enforcing both value and type of `a` to be the same as that of `b`.

Existential types do not work with:

- Class instances to be hidden together with the existential.

3.7 Kind inference and signatures

Unknown kinds of type constructors do not default to kind `*`, instead they lead to polymorphic kinds:

```
data Eq a b = Eq (forall f . f a -> f b)        -- Eq :: Forall a . a -> a -> *
```

If a more restrictive kind for `Eq` is desired, this can be enforced by a kind signature, similar to type signatures for values:

```
Eq :: * -> * -> *
```

3.8 Local instances

Instances can be declared locally:

```
eqInt :: Int -> Int -> Bool
eqInt = (==)

main :: IO ()
```

```
main
  = do let i = 3 :: Int
           j = 5 :: Int
       putStrLn (show (i == j))
       putStrLn (show (let m = 2 :: Int
                           instance Eq Int where
                             x == y = eqInt (x `mod` m) (y `mod` m)
                       in  i == j
                      ))
```

The above example respectively prints `False` and `True`.

Instance declarations as well as arising class constraints have a scope associated. The `Eq` constraint for the second `i == j` arose in the context of the local instance for `Eq Int`, which was then used for the context resolution in preference over the one defined globally in the Prelude.

During context resolution the following rules are used:

- Whenever a choice between instances is possible, the one with the innermost scope in relation to the to be proven constraint is chosen. If no such choice is possible (because scopes are equal) this is considered an overlapping instance.
- Class constraints over which is quantified lose their scope. They can arise in different scopes as part of instantiation in such a different scope. Quantified class constraints are not reduced via instances, only via superclasses, and only when the superclass constraint also arose. This is to avoid too early binding associated with a closed world of instances.
- Being in scope is static, that is, solely determined by the structure of the program text.

The implementation still has some quirks when abstracted dictionaries are involved. Sticking to ground instances like in the examples avoid this.

3.9 Lexically scoped type variables

Lexically scoped type variables can be introduced via

- pattern type signatures on arguments, for example in the following result type and type of an intermediate computation are connected:

  ```
  z (Wrap x) :: (mt,...) = let (m::mt,y) = properFraction x in (m::mt, Wrap y)
  ```

- pattern type signatures on a function result, as in the following:

  ```
  z a b c :: x = ... :: x
  ```

- type variables occurring in instance declarations, as in the following for `a` and `b`:

  ```
  instance (Sup (a (b ())) (a (b c)), Widen (a ()) (a (b ()))) => Sup (a ()) (a (b c)) where
     downcast o = case widen o :: Maybe (a (b ())) of
                    Just r   -> downcast r
                    Nothing  -> Nothing
  ```

3.10 Extensible records

Code generation for extensible records is not supported, hence no documentation is currently available.

3.11 Relaxed monomorphism restriction

UHC does feature a relaxed form of the monomorphism restriction: only bindings to non-variable pattern are enforced to be monomorphic. So, for example, for variable bindings like

```
x = 1
```

the type `Num a => a` is inferred for `x`. This overloading and loss of sharing can be avoided by using a type signature:

```
x :: Int
x = 1
```

or

```
x = (1 :: Int)
```

Monomorphism however is enforced for bindings to a pattern:

```
x1 = [1,2]          -- :: Num a => [a]
x2@(x2a:_) = x1     -- :: [Integer]
```

A first reason for monomorphism currently is pragmatic because in principle the definition of `x2a` is equivalent to:

```
x2a = head x1               -- :: Num a => a
```

That is, the need for a `Num a` propagates to fields of an aggregrate value. Automatic propagation of these required predicates is not done automatically in UHC, hence the monomorphism restriction in this case.

Potentially more subtly problematic however are overloaded bindings inside a tuple (or other aggregrates):

```
newtype Wrap = Wrap Double

-- properFraction :: (RealFrac a,Integral b) => a -> (b,a)

z (Wrap x) = let my@(m,y) = properFraction x in (m, Wrap y)
```

If generalized on their own, both `m` and `y` would be generalized and overloaded, losing any typing connection/constraint between them, which in turn leads to overgeneralization, which leads to unexpected ambiguity.

On its own the non-monomorphically restricted `m` would have the following type:

```
m :: Integral a => a
```

This could be fixed by using lexically scoped type variables (page 15):

```
z (Wrap x) :: (mt,...) = let (m::mt,y) = properFraction x in (m::mt, Wrap y)
```

but the above is the second reason for enforcing monomorphism for the aggregrate value.

(In the above partial type signatures (page 12) were also used.)

3.12 Default for ambiguous overloading

Defaulting is implemented following mostly Haskell prime defaulting proposal 2, i.e. per class defaulting is allowed. One default can be defined per class, e.g. for class C:

```
class C a where
  c :: a -> a

instance C Int where
  c = id

default C Int
```

Interaction with multiple parameter type classed is unpredictable, and should be avoided. Defaults cannot be turned off (with `default ()`). The Haskell98 form of `default` is syntactically allowed, but silently ignored. Multiple types may be specified, currently only the first one is used:

```
default C (Int,Integer)
```

Scope of default definitions is global, that is, once imported somewhere on an import chain, always imported, thereby having the same behavior as instance declarations w.r.t. scope. This may change in the future.

The prelude defines

```
default Num Integer
default Real Integer
default Enum Integer
default Integral Integer
default Fractional Double
default RealFrac Double
default Floating Double
default RealFloat Double
```

3.13 Infix type constructors, classes, and type variables

UHC allows type constructors, classes, and type variables to be operators, and to be written infix, as does GHC. See (e.g.) GHC's documentation on Infix type constructors, classes, and type variables

3.14 Pragmas

UHC allows pragmas, wrapped inside special commentlike delimiters `{-#` and `#-}`, e.g.:

```
{-# LANGUAGE CPP #-}
```

The following pragmas are supported.

- `{-# LANGAUGE pragma #-}` (file header) pragmas, where pragma may be:
  - `CPP` : switch on cpp preprocessing. See also preprocessing (page 18).
  - `NoImplicitPrelude`: don't automatically import `Prelude` and/or assume its presence.
  - `GenericDeriving`, `NoGenericDeriving`: turn on/off respectively generic deriving, default is on.
  - `BangPatterns`, `NoBangPatterns`: turn on/off respectively bang patterns, default is on.
  - `PolyKinds`, `NoPolyKinds`: turn on/off respectively polymorphic kind inference, default is off.

  - `OverloadedStrings`, `NoOverloadedStrings`: turn on/off overloaded strings, default is off. See also GHC doc. The (current) difference is that the module `Data.String` is brought in scope, although qualified, and only for `fromString`.
  - `ExtensibleRecords`: turn on special syntax for extensible records. Available for internal backwards compatibility, but otherwise useless as codegeneration and runtime currently does not support extensible records. It reserves the operators `#` and `:=` for record selection and update respectively.

- `{-# DERIVING class field generic-function #-}` pragma: see generic deriving (page 11).
- `{-# EXCLUDE_IF_TARGET targets #-}` (file header pragma) make the module invisible for compilation. This is a (hopefully) temporary measure to deal with the abilities of distinctive backend. For example, the `js` (Javascript) backend does not support (e.g.) file access.
- `{-# OPTIONS_UHC "..." #-}` (file header pragma) provide compiler options as a string to be parsed as if it were passed via the commandline.

Other or unsupported pragmas are silently ignored, even when appropriate currently no warning will be given. The text inside the pragma delimiters is then treated as normal comment, as were it inside plain `{-` and `-}` comment delimiters.

3.15 Preprocessing (with `cpp`)

With `LANGUAGE` pragma `CPP` defined, or option `--cpp` specified, source text will be preprocessed by `cpp` before compilation. With option `--optP` additional options passed to `cpp` may be given.

The following compile time flags are then defined. Unless mentioned otherwise the flags are just defined, i.e. have no value.

- `__UHC__`, which has the numerical form of the compiler version as its value (See also option `--version-asnumber`).
- `__UHC_TARGET_X__`, where `X` stands for the current backend target.
- `__UHC_BUILDS_X__`, where `X` stands for
  - `O` when `C` compilation is done (only relevant for C code compilation, but `cpp` is then invoked as part of `C` compilation).
  - `CPP` when `cpp` preprocessing is done.

4 Haskell compatibility

4.1 Haskell 98

The following Haskell98 features are not or partially supported:

- N+k patterns are not implemented.
- Bound variables in an irrefutable pattern all are individually irrefutable, nested irrefutability is ignored.
- Literate Haskell allows `\begin{code}` .. `\end{code}`, but not individual lines prefixed with `>` .
- Strictness annotation in datatype definitions is allowed but ignored.
- The monomorphism restriction is implemented in a relaxed variant.

4.2 Haskell 2010

The following Haskell 2010 features are not or partially supported:

- The foreign function interface (See ForeignFunctionInterface) is implemented to the point where base libraries can be constructed. In particular `export` is not implemented, and neither are `dynamic` and `wrapper` imports. The `stdcall` calling convention is not implemented.
- Some, but not all `LANGUAGE` pragmas (LanguagePragma) are recognised, others are ignored. See pragmas (page 17).
- Pattern guards (PatternGuards) are not implemented.

5 Backend specifics

5.1 Foreign function interface (FFI)

5.1.1 Primitives

A limited form of FFI is allowed, only to define primitives. A special calling convention `prim` is used to indicate such primitives. The calling convention follows the necessary calling convention for the backend for which code is generated. For example, the `Prelude` contains:

```
foreign import prim primAddInt       :: Int -> Int -> Int
```

5.1.2 `bc` and `C` backend

The `bc` or `C` based target uses the `ccall` calling convention; the `jazy` target uses `jazy`. To ensure the use of C calling convention use `ccall`, but support depends on the target. For example, for target `bc` the `Prelude` defines:

```
foreign import ccall "sinf"  primSinFloat   :: Float -> Float
```

The `Foreign.XX` library modules are available. Calling conventions `dynamic`, `wrapper`, `export` as well as wrapping as finalizers is not yet supported.

5.1.3 `js` backend

The `js` calling convention for the `js` (Javascript) backend allows import entities to specify a little bit of OO like notation and array access. The subsequent is taken from the UHC blog:

Haskell's Foreign Function Interface (FFI) predefined calling conventions do not match well with Javascript's object oriented features. In particular selecting a field of an object using dot notation (like `o.f`) and using an object as an array using brackets (like `o[i]`) do not have a natural counterpart in Haskell or the default calling conventions supported by the FFI interface. So, here are some examples of how Javascript is accessed in UHC via its `jscript` calling convention:

```
data Document
foreign import jscript "document"   document      :: IO Document
foreign import jscript "%1.write(%*)" documentWrite :: Document -> JSString -> IO ()
foreign import jscript alert :: JSString -> IO ()
```

From within a browser the document representation can be accessed via the global variable `document`, the foreign entity `"document"` translates to a reference to this variable. The type of the document is defined as an opaque type, it can thus only manipulated via Javascript. Writing a string to the

document is done by invoking the method `write` on a document. The foreign entity `"%1.write(%*)"` specifies that from all arguments the first one is used as the receiver of the method `write`. The parenthesis `()` specify that a call has to be made, where `%*` means passing all arguments except those referred to explicitly by means of `%<nr>`, where `<nr> >= 1` refers to argument `<nr>`. If an entity is omitted as in `alert` it defaults to `"<functionname>(%*)"` where `<functionname>` is the name of the foreign function being defined.

Function `documentWrite` does not accept a `String` but a `JSString` instead, defined to be the platform dependent representation of Strings, converted to and from `String` with corresponding conversion functions.

```
type JSString = PackedString
stringToJSString :: String -> JSString
jsStringToString :: JSString -> String
```

`stringToJSString` forces its argument to be fully evaluated and then converts it to a Javascript String.

There is choice whether to put `document` in the `IO` monad or not, depending whether this global object itself will ever be assigned a new value or not. Not being a Javascript DOM wizard wrapping in `IO` seems to be the safest bet.

Given these functions a minimal Hello World web program thus is:

```
main = alert $ stringToJSString "Hi there!"
```

As this would pop up an alert box, an alternative Hi is the following program which writes to the document instead:

```
main = do d <- document
          documentWrite d $ stringToJSString "Hi there!"
```

Actually, the usual Hello would have worked as well because it is implemented as writing to the document:

```
main = putStr "Hi there!"
```

To show the usefulness of array like access as part of we do a bit of rudimentary DOM programming:

```
foreign import jscript "%1.getElementsByName(%*)" documentGetElementsByName :: Document -> JSString -> IO (NodeList Nod

data NodeList x

foreign import jscript "%1.length" nodeListLength :: NodeList Node -> Int
foreign import jscript "%1[%2]"    nodeListItem   :: NodeList Node -> Int -> IO Node

data Node

foreign import jscript "%1.innerHTML" elementInnerHTML :: Node -> JSString
foreign import jscript "%1.tagName"   elementTagName   :: Node -> JSString
```

A `NodeList` is not an array, but behaves like an array: we can ask for its length and retrieve an element by index. It is not an array itself, so modelling it as such in Haskell would be incorrect. However, by allowing import entities to use Javascript array notation we circumvent this limitation and the Javascript array interface can still be used easily.

Finally, this minimal interface to DOM can be used to retrieve and print info about an element in an html document:

```
main = do d <- document
          nl <- documentGetElementsByName d (stringToJSString "myHeader")
          print (nodeListLength nl)
          n <- nodeListItem nl 0
          print $ jsStringToString $ elementTagName n
          print $ jsStringToString $ elementInnerHTML n
```

Given the presence of

```
<h1 name="myHeader">Head says hello!</h1>
```

with the name `"myHeader"` in the document where the program is run, it will produce the following as part of the document:

```
1 "H1" "Head says hello!"
```

For reference, import entities follow the following grammar:

```
exp    ::=  '{}'                        -- Haskell constructor to JS object
          |  (arg | ident) post *         -- JS expression
post   ::=  '.' ident                   -- object field
          |  '[' exp ']'                -- array indexing
          |  '(' args ')'               -- function call
args   ::=  epsilon | arg (, arg) *     -- possible arguments
arg    ::=  '%' ('*' | int)             -- all arguments, or a specific one
          |  '"' str '"'                -- literal text

ident  ::= a valid JavaScript identifier
int    ::= any integer
str    ::= any string
```

where `ident` is a Haskell like lowercase/uppercase identifier, and where parenthesis only may appear at the end.

See also The Utrecht Haskell Compiler JavaScript Backend Page

5.2 Limitations

6 Language implementation status

An older overview can be found on older pages on the first release and language features.

6.1 Included packages

The following packages are included and precompiled. When marked with version 1.0.0.0 it has no relationship with the 'real' (hackage) package providing just enough for package `haskell98`, other version numbers mean the same functionality is offered unless specified otherwise.

- `uhcbase`, version same as compiler version.
- `base`, version 3.0.0.0, which is arbitrary. The `C` backend only has a minimal `System.IO`, which propagates to the `haskell98` package.
- `array`, version 1.0.0.0
- `filepath`, version 1.1.0.4
- `oldlocale`, version 1.0.0.2

- `oldtime`, version 1.0.0.4
- `unix`, version 1.0.0.0
- `directory`, version 1.0.0.0
- `random`, version 1.0.0.2
- `haskell98`, version 1.0.1.1, lacking `System.Cmd(system)`, handicapped as described by `base` package.

6.2 Library modules

The library consists of

- UHC specific modules, tightly coupled with the underlying implementation. The names of these modules start with `UHC.`.
- Modules from the GHC distribution, used without changes.
- Modules cloned + adapted from the GHC distribution. This maybe involve a couple of `#ifdef`s or more extensive changes. The intention is to have these changes merged back into the GHC library as to avoid library differences between Haskell implementations.
- Modules taken from hackage.

In due time the intention is to build as much as possible with cabal, sharing as much as possible with hackage.

There are several known problems in the `uhcbase` and `base` package:

- `Data.Typeable`: `cast` (and variants) always yields `Nothing`. Equality of type representations is not done efficiently with hashing (not yet available). but with the usual structural comparison of the type representation.
- `Foreign`, `Ptr`, `Weak`, etc: garbage collection related functionality is influenced by the use of Boehm GC, e.g. only a minimal interface for weak pointers, finalizers, etc is available, not doing very much.
- Using functions that works with the internal structure of a handle (eg. `Prelude.print` inside a weak pointer finalizer will cause the program to crash.
- Nested calls of the internal function `UHC.Handle.withHandle` will cause the program to fail.

6.3 Library + runtime

The C based backends (C, bc) use the following public domain libraries:

6.3.1 Third party libraries

- LibTomMath has been cloned and adapted to play nicely with the garbage collector.

6.4 Known issues

- Stacksize for the C backend is fixed and not checked for overflow.
- Error messages involving types are difficult to decode; too much of the internals of the type system machinery is exposed.
- Compilation of a single module which uses many imports (directly or indirectly) takes noticably more time because reading in `.hi` files takes time.
- Executables for code generation target `bc` get quite large: the C encoding of bytecode files for modules is rather inefficient and linking together with the base libraries draws in most of the library code.
- Large expressions involving many class constraints take non lineair compile time, consumed by context reduction.
- When compiling with option `--cpp` or language pragma `CPP` the file preprocessed by cpp is put in the same directory as the source file. Without write permission this will fail.
- Overlapping instances are not reported as such, but indirectly by producing an error that a class constraint could not be proven.
- Local instances are not always working properly. Stick to its use without abstraction, i.e. only ground instances as in the examples (page 14).
- Regression testing may report differences because of different linefeed encoding on different platforms.
- Instances for tuples are currently limited to at most 5-tuples. Also not all standard classes have definitions for tuples, in particular `Read`.
- Resulting code does not execute fast. Yes we know, many 'standard' optimizations are not yet implemented, and the whole program analysis backends require those optimizations as well to be effective.
- Whole program analysis takes a lot of time.

6.5 Fully functional backends

6.5.1 `bc`: GRIN based interpreter, no whole program analysis

This is the default backend, and the most complete.

- The size of the stack is hardcoded, overflow is runtime checked.

6.6 Almost fully functional backends

6.6.1 `C`: GRIN based C, with whole program analysis

- The HPT analysis is sensitive to particular generated code constructs, especially when related to foreign functions. Some libraries, when used, therefore make compilation fail.
- No exceptions, throwing an exception will stop the program.
- Runs simple programs using the Prelude only; primitives for many of the other libraries are not yet implemented. Not all modules available for the `bc` backend are supported, in particular those interacting with the underlying system: IO, CPUTime, ...
- The size of the stack is hardcoded. No runtime overflow check is done.

6.6.2 `jazy`: Core based Java, no whole program analysis

- Only when enabled via `configure --enable-java`.
- Runs for latest variant, but partial implementation of required primitives.
- No exceptions.
- Jar files and created .class files all together get big. Code is not causing this, but the tables holding constants referring to other classes. JVM expects this to be done per .class file. Because all functions and CAFs have a separate class definition this becomes quite costly in terms of jar file size and runtime startup time.

6.6.3 `js`: Core based JavaScript, no whole program analysis

- Available by default, no configure flag to turn it off/on.
- Runs for latest variant, but all FFI related to the usual OS stuff is not available.
- No exceptions (yet).
- No cabal support (yet).

6.7 Partially functional backends

6.7.1 `llvm`: GRIN based LLVM, with whole program analysis

- For variant 8 only, little runtime support.

6.7.2 `clr`: GRIN based CLR, with whole program analysis

- For variant 8 only, no runtime support.
- Only when enabled via `configure --enable-clr`.

7 Troubleshooting FAQ

7.1 The compiler seems to loop, or complains about premature end of file, or complains with "ehc: Prelude.chr: bad argument: 4087767"

Remove `.hi` files manually, as it seems not to be possible to guarantee that `.hi` files from a previous version, or another haskell compiler, are detected as invalid.

7.2 I have installed UHC previously, now the libraries do not compile

From a previous install the library path settings may linger around. This information is stored in the file reported by:

```
% uhc --meta-dir-env
/Volumes/Work/.uhc-101
%
```

Remove this directory. From version 1.0.1 onwards this should no longer be a problem as the version number is encoded in the filename:

```
% uhc --meta-dir-env
/Volumes/Work/.uhc-1.0.1-101
%
```

From version 1.1 onwards the above mechanism is not used anymore.

7.3 I have checked out a new EHC version over a previously working one, and now it does not compile anymore

Many things currently change, in particular also in the build process, so when checking out a new version over an old one old stuff may linger around. If you get build errors for a `<variant>` try to fix it with one the following, in order of severity:

- Rerun `./configure`.
- Clean with `make <variant>/clean`.
- Manually remove the build directory (`EHCHOME/build`) and the installation directory for building (`EHCHOME/install-for-build`).

If this does not fix the building, send us the full build output.

8 License

```
The Utrecht Haskell Compiler (UHC) License
==========================================

UHC follows the advertisement free BSD license, of which the basic
template can be found here:

        http://www.opensource.org/licenses/bsd-license.php

UHC uses the following libraries with their own license:
- Library code from the GHC distribution, see comment in the modules in ehclib

License text
============

Copyright (c) 2009-2010, Utrecht University, Department of Information
and Computing Sciences, Software Technology group

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
```

```
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

9 Further reading

See also

- How to experiment with EHC.
- Structure of EHC.
- Internal (technical) documention of EHC.
- Shuffle for manipulating source chunks.
- Text2Text for documentation formatting.
- GHC.
- HUT library.
- The Utrecht Haskell Compiler JavaScript Backend Page.